第八节:spring 事物源码解析

一:事物概念解析

1.1)**什么是事物**?

事务是逻辑上的一组执行单元，要么都执行，要么都不执行.

1.2)**事物的特性**(ACID)

什么是ACID

ACID是指数据库管理系统DBMS中事物所具有四个特性

eg:在数据库系统中，一个事务由一系列的数据库操作组成一个完整的逻辑过程，比如银行转账，从原账户扣除金额，目标账户增加金额

①:atomicity【**原子性**】

原子性表现为**操作不能被分割**，那么这二个操作 要么**同时完成**，要么就**全部不完成**，若事务出错了，那么事务就会回滚，

好像什么 都 没有发生过

②:Consistency【**一致性**】

一致性也比较容易理解，也就是说数据库要一直处于一致的状态，事务开始前是一个一致状态，事务结束后是另一个一致状态，

事务将数据库从一个一致状态转移到另一个一致状态

③:Isolation【**隔离性**】

所谓的独立性就是指并发的事务之间不会互相影响，如果一个事务要访问的数据正在被另外一个事务修改，只要另外一个事务还未提交，它所访问的数据就不受未提交事务的影响。换句话说，一个事务的影响在该事务提交前对其它事务是不可见的

④:Durability【**持久性**】

若事务已经提交了，那么就回在数据库中永久的保存下来

二:Spring事务三大接口介绍

2.1）PlatformTransactionManager： **（平台）事务管理器**

2.2）TransactionDefinition： **事务定义信息(事务隔离级别、传播行为、超时、只读、回滚规则**

2.3）TransactionStatus： **事务运行状态**

2.1)PlatformTransactionManager接口介绍

Spring**并不直接管理事务，而是提供了多种事务管理器** ，他们将事务管理的职责委托给Hibernate或者JTA等持久化机制所提供的相关平台框架的事务来实现。

Spring事务管理器的接口是:

org.springframework.transaction.PlatformTransactionManager ，
通过这个接口，Spring为各个平台如JDBC、Hibernate等都提供了对应的事务管理器，但是具体的实现就是各个平台自己的事情了。

```
public interface PlatformTransactionManager {
 /**
 *获取事物状态
 */
 TransactionStatus getTransaction(@Nullable TransactionDefinition definition) throws TransactionException;
 /**
 *事物提交
 */
 void commit(TransactionStatus status) throws TransactionException;
 /**
 *事物回滚
 */
 void rollback(TransactionStatus status) throws TransactionException;
}
```

2.2)TransactionDefinition 事物属性的定义

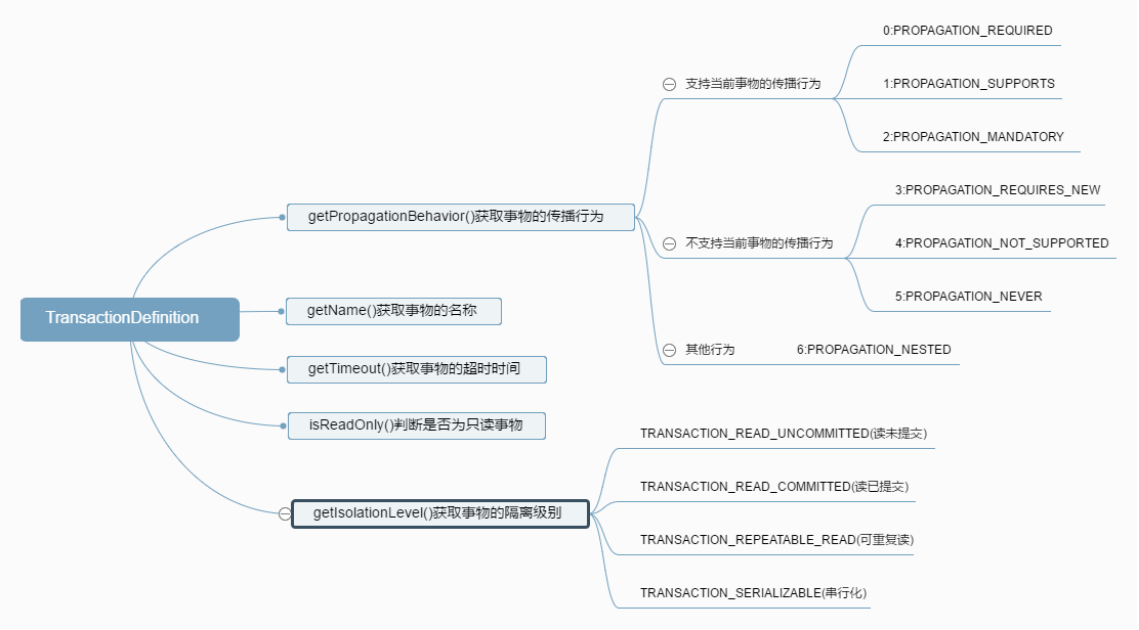
org.springframework.transaction.TransactionDefinition

TransactionDefinition接口中定义了5个方法以及一些表示事务属性的常量比如隔离级别、传播行为等等的常量。
我下面只是列出了TransactionDefinition接口中的方法而没有给出接口中定义的常量，该接口中的常量信息会在后面依次介绍到

```
public interface TransactionDefinition {

      /**
       * 支持当前事物，若当前没有事物就创建一个事物
       * */
      int PROPAGATION_REQUIRED = 0;

   /**
    *  如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行
    * */
      int PROPAGATION_SUPPORTS = 1;

      /**
    *如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常
       */
      int PROPAGATION_MANDATORY = 2;

   /**
    *创建一个新的事务，如果当前存在事务，则把当前事务挂起
    **/
      int PROPAGATION_REQUIRES_NEW = 3;

   /**
    *  以非事务方式运行，如果当前存在事务，则把当前事务挂起
    * */
      int PROPAGATION_NOT_SUPPORTED = 4;

   /**
    * 以非事务方式运行，如果当前存在事务，则抛出异常。
    * */
      int PROPAGATION_NEVER = 5;
```

```
    /**
     *  表示如果当前正有一个事务在运行中，则该方法应该运行在 一个嵌套的事务中，
       被嵌套的事务可以独立于封装事务进行提交或者回滚(保存点)，
       如果封装事务不存在,行为就像 PROPAGATION_REQUIRES NEW
     * */
       int PROPAGATION_NESTED = 6;


       /**
     *使用后端数据库默认的隔离级别，Mysql 默认采用的 REPEATABLE_READ隔离级别 Oracle 默认采用的 READ_COMMITTED隔离级
       */
       int ISOLATION_DEFAULT = -1;

       /**
     *最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读
       */
       int ISOLATION_READ_UNCOMMITTED = Connection.TRANSACTION_READ_UNCOMMITTED;

       /**
     *允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生
       */
       int ISOLATION_READ_COMMITTED = Connection.TRANSACTION_READ_COMMITTED;

       /**
     *对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生
       */
       int ISOLATION_REPEATABLE_READ = Connection.TRANSACTION_REPEATABLE_READ;

       /**
        *最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，
        * 也就是说，该级别可以防止脏读、不可重复读以及幻读。但是这将严重影响程序的性能通常情况下也不会用到该级别
        */
       int ISOLATION_SERIALIZABLE = Connection.TRANSACTION_SERIALIZABLE;


       /**
     *使用默认的超时时间
       */
       int TIMEOUT_DEFAULT = -1;


       /**
     *获取事物的传播行为
       */
       int getPropagationBehavior();

       /**
     *获取事物的隔离级别
       */
       int getIsolationLevel();

       /**
     *返回事物的超时时间
       */
       int getTimeout();

       /**
     *返回当前是否为只读事物
       */
       boolean isReadOnly();

       /**
```

```
    *获取事物的名称
     */
    @Nullable
    String getName();

}
```



2.3)TransactionStatus**接口介绍**

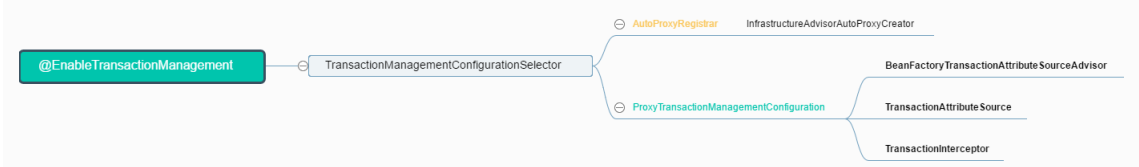
TransactionStatus接口用来记录事务的状态 该接口定义了一组方法,用来获取或判断事务的相应状态信息.

PlatformTransactionManager.getTransaction(...) 方法返回一个 TransactionStatus 对象。返回的 TransactionStatus 对象可能代表一个新的或已经存在的事务（如果在当前调用堆栈有一个符合条件的事物

```
public interface TransactionStatus extends SavepointManager, Flushable {

    /**
     * 是否为新事物
     * */
        boolean isNewTransaction();

        /**
     *是否有保存点
         */
        boolean hasSavepoint();

        /**
     *设置为只回滚
         */
        void setRollbackOnly();

        /**
     *是否为只回滚
         */
        boolean isRollbackOnly();

        /**
```

```
        *属性
        */
        @Override
        void flush();

        /**
     *判断当前事物是否已经完成
        */
        boolean isCompleted();

}
```

三:我们来分析@EnableTransactionManagement注解来给我们容器加入了什么组件



从源码开始分析注册的组件

3.1)@EnableTransactionManagement**开始分析**

```
org.springframework.transaction.annotation.EnableTransactionManagement

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import(TransactionManagementConfigurationSelector.class)
public @interface EnableTransactionManagement {

    /**
     * 指定使用什么代理模式(true为cglib代理,false 为jdk代理)
     * */
        boolean proxyTargetClass() default false;

    /**
     * 通知模式 是使用代理模式还是aspectj  我们一般使用Proxy
     * */
        AdviceMode mode() default AdviceMode.PROXY;

        int order() default Ordered.LOWEST_PRECEDENCE;
}
```

3.2)我们从3.1处的源码可以分析处他通过@Import导入了TransactionManagementConfigurationSelector组件

TransactionManagementConfigurationSelector**源码分析**

我们可以分析处向容器中导入了二个组件

1)AutoProxyRegistrar

2)ProxyTransactionManagementConfiguration

```
public class TransactionManagementConfigurationSelector extends AdviceModeImportSelector<EnableTransactionManage

    /**
     * 往容器中添加组件 1) AutoProxyRegistrar
     *                  2) ProxyTransactionManagementConfiguration
```

```
    * */
    @Override
    protected String[] selectImports(AdviceMode adviceMode) {
        switch (adviceMode) {  //因为我们配置的默认模式是PROXY
            case PROXY:
                return new String[] {AutoProxyRegistrar.class.getName(),
                        ProxyTransactionManagementConfiguration.class.getName()};
            case ASPECTJ:
                return new String[] {
                        TransactionManagementConfigUtils.TRANSACTION_ASPECT_CONFIGURATION_CLASS_NAM
            default:
                return null;
        }
    }

}
```

3.2.1)**首先我们来分析**AutoProxyRegistrar**给我们容器中干了什么?**

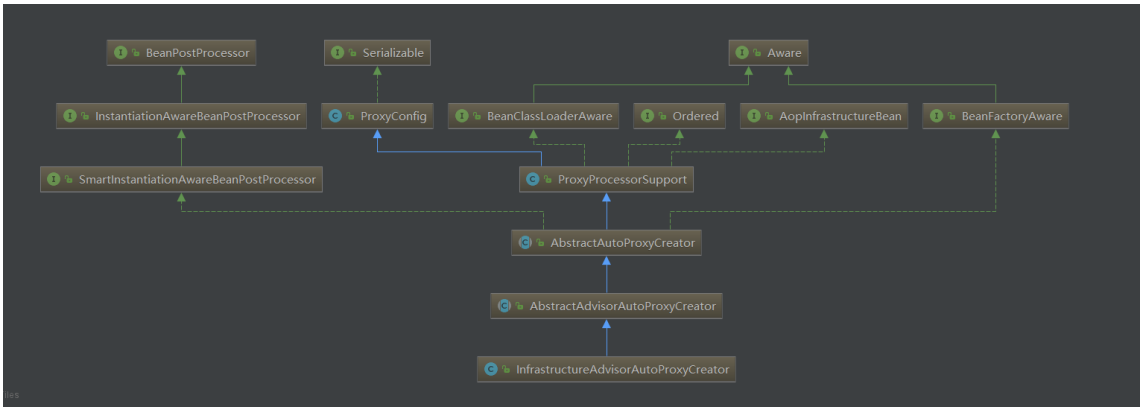
从源码分析出，AutoProxyRegistrar**为我们容器注册了一个**InfrastructureAdvisorAutoProxyCreator**组件**

```
public class AutoProxyRegistrar implements ImportBeanDefinitionRegistrar {

    private final Log logger = LogFactory.getLog(getClass());

    @Override
    public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {
        boolean candidateFound = false;
        //从我们传入进去的配置类上获取所有的注解的
        Set<String> annoTypes = importingClassMetadata.getAnnotationTypes();
        //循环我们上一步获取的注解
        for (String annoType : annoTypes) {
            //获取注解的元信息
            AnnotationAttributes candidate = AnnotationConfigUtils.attributesFor(importingClassMetadata, annoType
            if (candidate == null) {
                continue;
            }
            //获取注解的mode属性
            Object mode = candidate.get("mode");
            //获取注解的proxyTargetClass
            Object proxyTargetClass = candidate.get("proxyTargetClass");
            //根据mode和proxyTargetClass的判断来注册不同的组件
            if (mode != null && proxyTargetClass != null && AdviceMode.class == mode.getClass() &&
                    Boolean.class == proxyTargetClass.getClass()) {
                candidateFound = true;
                if (mode == AdviceMode.PROXY) {
                    AopConfigUtils.registerAutoProxyCreatorIfNecessary(registry);
                    if ((Boolean) proxyTargetClass) {
                        AopConfigUtils.forceAutoProxyCreatorToUseClassProxying(registry);
                        return;
                    }
                }
            }
        }
    }

}

    public static BeanDefinition registerAutoProxyCreatorIfNecessary(BeanDefinitionRegistry registry) {
        return registerAutoProxyCreatorIfNecessary(registry, null);
    }

    public static BeanDefinition registerAutoProxyCreatorIfNecessary(BeanDefinitionRegistry registry, Object source) {
```

```
        return registerOrEscalateApcAsRequired(InfrastructureAdvisorAutoProxyCreator.class, registry, source);
    }

    private static BeanDefinition registerOrEscalateApcAsRequired(Class<?> cls, BeanDefinitionRegistry registry, Object
        Assert.notNull(registry, "BeanDefinitionRegistry must not be null");

    //判断容器中有没有org.springframework.aop.config.internalAutoProxyCreator名字的bean定义,
        if (registry.containsBeanDefinition(AUTO_PROXY_CREATOR_BEAN_NAME)) {
            BeanDefinition apcDefinition = registry.getBeanDefinition(AUTO_PROXY_CREATOR_BEAN_NAME);
            if (!cls.getName().equals(apcDefinition.getBeanClassName())) {
                int currentPriority = findPriorityForClass(apcDefinition.getBeanClassName());
                int requiredPriority = findPriorityForClass(cls);
                if (currentPriority < requiredPriority) {
                    apcDefinition.setBeanClassName(cls.getName());
                }
            }
            return null;
        }


    //自己注册一个org.springframework.aop.config.internalAutoProxyCreator的组件
        RootBeanDefinition beanDefinition = new RootBeanDefinition(cls);
        beanDefinition.setSource(source);
        beanDefinition.getPropertyValues().add("order", Ordered.HIGHEST_PRECEDENCE);
        beanDefinition.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
        registry.registerBeanDefinition(AUTO_PROXY_CREATOR_BEAN_NAME, beanDefinition);
        return beanDefinition;
    }
```

3.2.1.1)**我们从**3.2.1**可以分析出**AutoProxyRegistrar**会为我们容器中导入了一个叫**InfrastructureAdvisorAutoProxyCreator**的组件**

①:我们来看下 InfrastructureAdvisorAutoProxyCreator**继承图,有没有一点熟悉的味道**?



所以我们来分析InfrastructureAdvisorAutoProxyCreator **实现了如下的接口**

①:实现了 Aware**接口(具体代表** BeanFactoryAware**接口)**

做了什么事情)

a：**把我们的**BeanFacotry**容器设置到了**InfrastructureAdvisorAutoProxyCreator**组件中去**

b: **创建了一个**advisorRetrievalHelper**组件　增强器检索工具**

AbstractAutoProxyCreator 实现了BeanFactoryAware接口，但是马上又被AbstractAdvisorAutoProxyCreator给重写了;

```
org.springframework.aop.framework.autoproxy.AbstractAutoProxyCreator
public void setBeanFactory(BeanFactory beanFactory) {
    this.beanFactory = beanFactory;
}
马上又被org.springframework.aop.framework.autoproxy.AbstractAdvisorAutoProxyCreator重写了
public void setBeanFactory(BeanFactory beanFactory) {
    super.setBeanFactory(beanFactory);
```

```
        if (!(beanFactory instanceof ConfigurableListableBeanFactory)) {
            throw new IllegalArgumentException(
                    "AdvisorAutoProxyCreator requires a ConfigurableListableBeanFactory: " + beanFactory);
        }
        initBeanFactory((ConfigurableListableBeanFactory) beanFactory);
    }

    protected void initBeanFactory(ConfigurableListableBeanFactory beanFactory) {
        this.advisorRetrievalHelper = new BeanFactoryAdvisorRetrievalHelperAdapter(beanFactory);
    }

    但是AbstractAdvisorAutoProxyCreator类的initBeanFactory又被InfrastructureAdvisorAutoProxyCreator重写了
    org.springframework.aop.framework.autoproxy.InfrastructureAdvisorAutoProxyCreator
    protected void initBeanFactory(ConfigurableListableBeanFactory beanFactory) {
        super.initBeanFactory(beanFactory);
        this.beanFactory = beanFactory;
    }
```

②:实现了我们的接口 InstantiationAwareBeanPostProcessor类型的后置处理器，为我们容器中做了什么事情

org.springframework.aop.framework.autoproxy.AbstractAutoProxyCreator#postProcessBeforeInitializa

postProcessBeforeInstantiation**方法**

```
    public Object postProcessBeforeInstantiation(Class<?> beanClass, String beanName) throws BeansException {
        Object cacheKey = getCacheKey(beanClass, beanName);

        if (beanName == null || !this.targetSourcedBeans.contains(beanName)) {
            if (this.advisedBeans.containsKey(cacheKey)) {
                return null;
            }
            if (isInfrastructureClass(beanClass) || shouldSkip(beanClass, beanName)) {
                this.advisedBeans.put(cacheKey, Boolean.FALSE);
                return null;
            }
        }

        // Create proxy here if we have a custom TargetSource.
        // Suppresses unnecessary default instantiation of the target bean:
        // The TargetSource will handle target instances in a custom fashion.
        if (beanName != null) {
            TargetSource targetSource = getCustomTargetSource(beanClass, beanName);
            if (targetSource != null) {
                this.targetSourcedBeans.add(beanName);
                Object[] specificInterceptors = getAdvicesAndAdvisorsForBean(beanClass, beanName, targetSource);
                Object proxy = createProxy(beanClass, beanName, specificInterceptors, targetSource);
                this.proxyTypes.put(cacheKey, proxy.getClass());
                return proxy;
            }
        }

        return null;
    }
```

postProcessAfterInstantiation**方法（没有做任何事情，直接返回）**

```
    public boolean postProcessAfterInstantiation(Object bean, String beanName) {
        return true;
    }
```

③:实现了我们的接口BeanPostProcessor类型的后置处理器，为我们容器中做了什么事情

postProcessBeforeInitialization**方法没有做任何事情,直接返回**

```
    public Object postProcessBeforeInitialization(Object bean, String beanName) {
        return bean;
    }
```

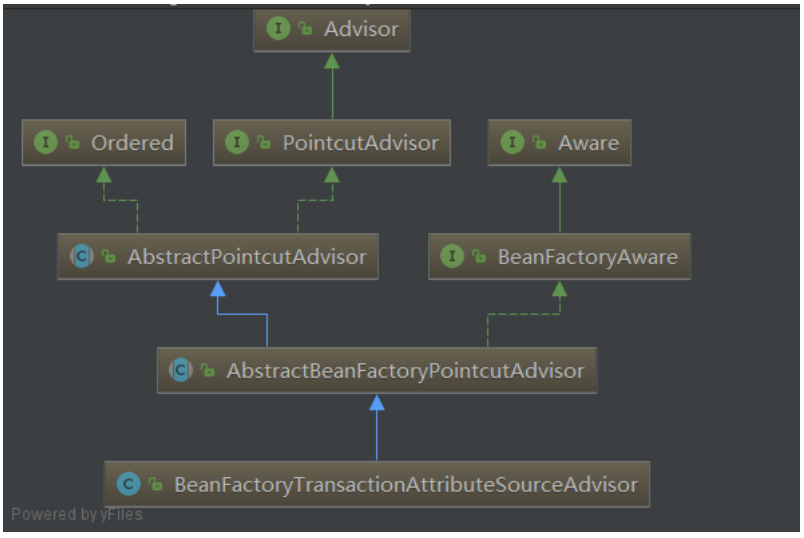
postProcessAfterInitialization **为我们做了事情**

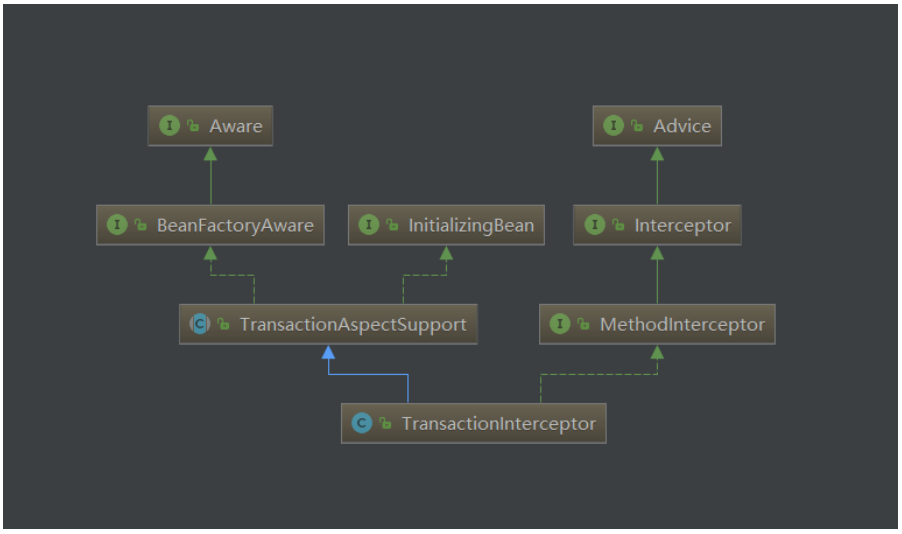
```
    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
        if (bean != null) {
            Object cacheKey = getCacheKey(bean.getClass(), beanName);
            if (!this.earlyProxyReferences.contains(cacheKey)) {
                return wrapIfNecessary(bean, beanName, cacheKey);
            }
        }
        return bean;
    }
```

3.3)**然后我们在来分析一下** TransactionManagementConfigurationSelector **为我们还导入了一个类** ProxyTransactionManagementConfiguration

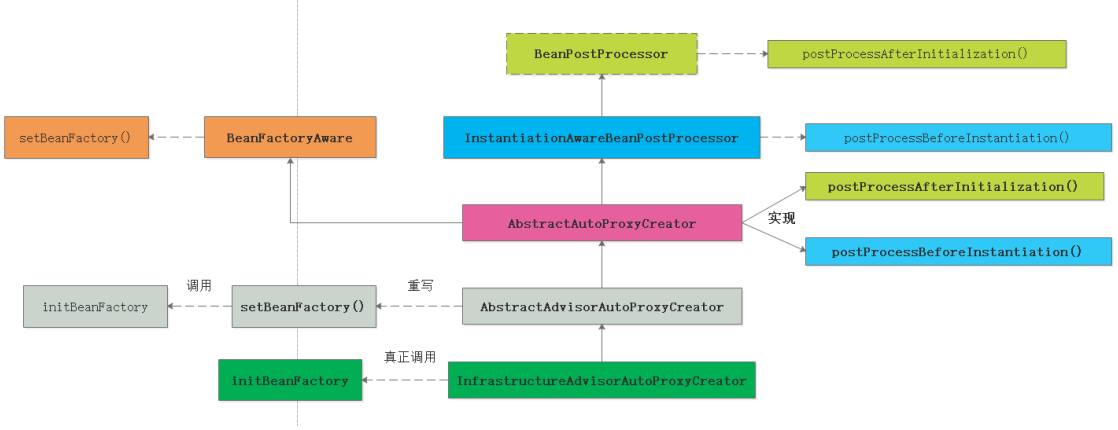
```
@Configuration
public class ProxyTransactionManagementConfiguration extends AbstractTransactionManagementConfiguration {


    /**
     * 为我我们容器中导入了 beanName为org.springframework.transaction.config.internalTransactionAdvisor
     * 类型为:BeanFactoryTransactionAttributeSourceAdvisor 的增强器
     *
     * */
    @Bean(name = TransactionManagementConfigUtils.TRANSACTION_ADVISOR_BEAN_NAME)
    @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
    public BeanFactoryTransactionAttributeSourceAdvisor transactionAdvisor() {
        BeanFactoryTransactionAttributeSourceAdvisor advisor = new BeanFactoryTransactionAttributeSourceAdvisor();
        //设置了事物源属性对象
        advisor.setTransactionAttributeSource(transactionAttributeSource());
        //设置了事物拦截器对象
        advisor.setAdvice(transactionInterceptor());
        advisor.setOrder(this.enableTx.<Integer>getNumber("order"));
        return advisor;
    }

    /**
     * 定义了一个事物属性源对象
     * */
    @Bean
    @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
    public TransactionAttributeSource transactionAttributeSource() {
        return new AnnotationTransactionAttributeSource();
    }

    /**
     * 事物拦截器对象
     *
     * */
    @Bean
    @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
    public TransactionInterceptor transactionInterceptor() {
        TransactionInterceptor interceptor = new TransactionInterceptor();
```

```
             * 把事物属性源对象设置到我们的事物拦截器对象中
            interceptor.setTransactionAttributeSource(transactionAttributeSource());
            //把我们容器中的 事物对象配置到事物拦截器中
            if (this.txManager != null) {
                  interceptor.setTransactionManager(this.txManager);
            }
            return interceptor;
      }

}
```





上诉我们画图总结，下面再将具体的源码分析



四:事物源代码解析流程

4.1)创建源代码过程

我们知道上图分析出，事物创建代理对象最最最主要的是InfrastructureAdvisorAutoProxyCreator这个类型作为
后置处理器为我们创建代理对象，实际上是他的父类AbstractAutoProxyCreator实现了postProcessBeforeInstantiation这个接口
org.springframework.aop.framework.autoproxy.AbstractAutoProxyCreator#postProcessBeforeInstantia
我们分析代码得出，再InstantiationAwareBeanPostProcessor.postProcessBeforeInstantiation没有为我们做了什么事情，那么是怎么创建代理对象的了？？？

```
public Object postProcessBeforeInstantiation(Class<?> beanClass, String beanName) throws BeansException {
        Object cacheKey = getCacheKey(beanClass, beanName);

        //判断我们的beanName以及是否处理过
        if (beanName == null || !this.targetSourcedBeans.contains(beanName)) {
            if (this.advisedBeans.containsKey(cacheKey)) {
                return null;
            }
            /*
             *判断当前的bean是不是基础的bean或者直接跳过，不需要代理的
               advice
                Pointcut
                Advisor
                AopInfrastructureBean
             **/
            if (isInfrastructureClass(beanClass) || shouldSkip(beanClass, beanName)) {
                this.advisedBeans.put(cacheKey, Boolean.FALSE);
                return null;
            }
        }

    /**
     * 判断我们容器中有没有自定义的targetSource 有为我们自动创建对象
     * 当时这一步的要求比较高，而且我们正常不会这里创建对象 ...
     * */
        if (beanName != null) {
            TargetSource targetSource = getCustomTargetSource(beanClass, beanName);
            if (targetSource != null) {
                this.targetSourcedBeans.add(beanName);
                Object[] specificInterceptors = getAdvicesAndAdvisorsForBean(beanClass, beanName, targetSource);
                Object proxy = createProxy(beanClass, beanName, specificInterceptors, targetSource);
                this.proxyTypes.put(cacheKey, proxy.getClass());
                return proxy;
            }
        }

        return null;
    }

    protected TargetSource getCustomTargetSource(Class<?> beanClass, String beanName) {
       //容器中必须要包含有个TargetSourceCreators 并且我们的组件也需要实现TargetSource接口
        if (this.customTargetSourceCreators != null &&
                this.beanFactory != null && this.beanFactory.containsBean(beanName)) {
            for (TargetSourceCreator tsc : this.customTargetSourceCreators) {
                TargetSource ts = tsc.getTargetSource(beanClass, beanName);
                if (ts != null) {
                    // Found a matching TargetSource.
                    if (logger.isDebugEnabled()) {
                        logger.debug("TargetSourceCreator [" + tsc +
                                "] found custom TargetSource for bean with name '" + beanName + "'");
```

```
                    }
                    return ts;
                }
            }
        }
        return null;
    }
```

4.2)我们知道上图分析出，事物创建代理对象最最最主要的是InfrastructureAdvisorAutoProxyCreator这个类型作为后置处理器为我们创建代理对象，实际上是他的父类AbstractAutoProxyCreator实现了 postProcessAfterInitialization这个接口

org.springframework.aop.framework.autoproxy.AbstractAutoProxyCreator#postProcessAfterInitialization

```
    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
        if (bean != null) {
            Object cacheKey = getCacheKey(bean.getClass(), beanName);
            if (!this.earlyProxyReferences.contains(cacheKey)) {
                //当前对象是否需要包装
                return wrapIfNecessary(bean, beanName, cacheKey);
            }
        }
        return bean;
    }


    protected Object wrapIfNecessary(Object bean, String beanName, Object cacheKey) {
        //判断代理对象再postProcessAfterInitialization接口中是否被处理过
        if (beanName != null && this.targetSourcedBeans.contains(beanName)) {
            return bean;
        }
        if (Boolean.FALSE.equals(this.advisedBeans.get(cacheKey))) {
            return bean;
        }
        //是否为基础的Bean 或者该对象不应该被调用
        if (isInfrastructureClass(bean.getClass()) || shouldSkip(bean.getClass(), beanName)) {
            this.advisedBeans.put(cacheKey, Boolean.FALSE);
            return bean;
        }

        //找到我们容器中所有的增强器
        Object[] specificInterceptors = getAdvicesAndAdvisorsForBean(bean.getClass(), beanName, null);

        //增强器不为空
        if (specificInterceptors != DO_NOT_PROXY) {
            this.advisedBeans.put(cacheKey, Boolean.TRUE);
            //创建代理对象
            Object proxy = createProxy(
                    bean.getClass(), beanName, specificInterceptors, new SingletonTargetSource(bean));
            this.proxyTypes.put(cacheKey, proxy.getClass());
            return proxy;
        }

        this.advisedBeans.put(cacheKey, Boolean.FALSE);
        return bean;
    }
```

```
org.springframework.aop.framework.autoproxy.AbstractAdvisorAutoProxyCreator#getAdvicesAndAdvisorsForBean(Abstra
    protected Object[] getAdvicesAndAdvisorsForBean(Class<?> beanClass, String beanName, TargetSource targetSource
        //找到合适的增强器
        List<Advisor> advisors = findEligibleAdvisors(beanClass, beanName);
        //增强器为空,不需要代理
        if (advisors.isEmpty()) {
            return DO_NOT_PROXY;
        }
        //返回增强器
        return advisors.toArray();
    }

/*
 *找到合适的增强器
 **/
protected List<Advisor> findEligibleAdvisors(Class<?> beanClass, String beanName) {
    //找到候选的增强器
        List<Advisor> candidateAdvisors = findCandidateAdvisors();
        //从候选的中挑选出合适的增强器
        List<Advisor> eligibleAdvisors = findAdvisorsThatCanApply(candidateAdvisors, beanClass, beanName);
        //增强器进行扩展
        extendAdvisors(eligibleAdvisors);
        //对增强器进行排序
        if (!eligibleAdvisors.isEmpty()) {
            eligibleAdvisors = sortAdvisors(eligibleAdvisors);
        }
        return eligibleAdvisors;
    }

==========================================findCandidateAdvisors();==================
/**
 * 找到候选的增强器
 * */
protected List<Advisor> findCandidateAdvisors() {
    //通过我们我们增强器探测工具找
        return this.advisorRetrievalHelper.findAdvisorBeans();
}

org.springframework.aop.framework.autoproxy.BeanFactoryAdvisorRetrievalHelper#findAdvisorBeans
public List<Advisor> findAdvisorBeans() {
        //看我们类级别缓存中有没有
        String[] advisorNames = this.cachedAdvisorBeanNames;
        if (advisorNames == null) {
            //去容器中查找实现了我们Advisor接口的实现类 的名称:(org.springframework.transaction.config.internalTransac
            advisorNames = BeanFactoryUtils.beanNamesForTypeIncludingAncestors(
                    this.beanFactory, Advisor.class, true, false);
            //放入到缓存中
            this.cachedAdvisorBeanNames = advisorNames;
        }
        if (advisorNames.length == 0) {
            return new ArrayList<Advisor>();
        }

        List<Advisor> advisors = new ArrayList<Advisor>();
        //循环我们的增强器
        for (String name : advisorNames) {
            //判断是不是合适的
            if (isEligibleBean(name)) {
                //当前的增强器是不是正在创建的
                if (this.beanFactory.isCurrentlyInCreation(name)) {
                    if (logger.isDebugEnabled()) {
```

```
                                logger.debug("Skipping currently created advisor '" + name + "'");
                            }
                        }
                        else {
                            try {
                               //通过getBean的显示调用获取BeanFactoryTransactionAttributeSourceAdvisor 组件
                                advisors.add(this.beanFactory.getBean(name, Advisor.class));
                            }
                            catch (BeanCreationException ex) {
                                Throwable rootCause = ex.getMostSpecificCause();
                                if (rootCause instanceof BeanCurrentlyInCreationException) {
                                    BeanCreationException bce = (BeanCreationException) rootCause;
                                    if (this.beanFactory.isCurrentlyInCreation(bce.getBeanName())) {
                                        if (logger.isDebugEnabled()) {
                                            logger.debug("Skipping advisor '" + name +
                                                    "' with dependency on currently created bean: " + ex.getMessage
                                        }
                                        // Ignore: indicates a reference back to the bean we're trying to advise.
                                        // We want to find advisors other than the currently created bean itself.
                                        continue;
                                    }
                                }
                                throw ex;
                            }
                        }
                    }
                }
                return advisors;
            }


            /**
             * 判断包含是否为合适的最终逻辑
             * 容器中的bean定义包含当前的增强器的bean定义，且bean的role是int ROLE_INFRASTRUCTURE = 2;
             * */
            protected boolean isEligibleAdvisorBean(String beanName) {
                return (this.beanFactory.containsBeanDefinition(beanName) &&
                        this.beanFactory.getBeanDefinition(beanName).getRole() == BeanDefinition.ROLE_INFRASTRUCTURE)
            }


========================================================= findAdvisorsThatCanApply(candidateAdvis
            protected List<Advisor> findAdvisorsThatCanApply(
                    List<Advisor> candidateAdvisors, Class<?> beanClass, String beanName) {

                ProxyCreationContext.setCurrentProxiedBeanName(beanName);
                try {
                   //真正的去候选的增强器中找到当前能用的增强器
                    return AopUtils.findAdvisorsThatCanApply(candidateAdvisors, beanClass);
                }
                finally {
                    ProxyCreationContext.setCurrentProxiedBeanName(null);
                }
            }
org.springframework.aop.support.AopUtils#findAdvisorsThatCanApply
            public static List<Advisor> findAdvisorsThatCanApply(List<Advisor> candidateAdvisors, Class<?> clazz) {
                //若传入进来的候选增强器为空直接返回
                if (candidateAdvisors.isEmpty()) {
                    return candidateAdvisors;
                }
                //创建一个本类能用的增前期集合
                List<Advisor> eligibleAdvisors = new LinkedList<Advisor>();
```

```
        //循环候选的增强器
        for (Advisor candidate : candidateAdvisors) {
            //判断增强器是不是实现了IntroductionAdvisor  很明显没实现该接口
            if (candidate instanceof IntroductionAdvisor && canApply(candidate, clazz)) {
                eligibleAdvisors.add(candidate);
            }
        }
        boolean hasIntroductions = !eligibleAdvisors.isEmpty();
        for (Advisor candidate : candidateAdvisors) {
            if (candidate instanceof IntroductionAdvisor) {
                // already processed
                continue;
            }
            //正在找出能用的增强器
            if (canApply(candidate, clazz, hasIntroductions)) {
                eligibleAdvisors.add(candidate);
            }
        }
        return eligibleAdvisors;
    }

    /**
     * 判断当前增强器是否为本来能用的
     * */
    public static boolean canApply(Advisor advisor, Class<?> targetClass, boolean hasIntroductions) {
        //根据类的继承图 发现 BeanFactoryTransactionAttributeSourceAdvisor没实现IntroductionAdvisor接口
        if (advisor instanceof IntroductionAdvisor) {
            return ((IntroductionAdvisor) advisor).getClassFilter().matches(targetClass);
        }
        //BeanFactoryTransactionAttributeSourceAdvisor实现了PointcutAdvisor接口
        else if (advisor instanceof PointcutAdvisor) {
           //强制转换为PointcutAdvisor
            PointcutAdvisor pca = (PointcutAdvisor) advisor;
            return canApply(pca.getPointcut(), targetClass, hasIntroductions);
        }
        else {
            // It doesn't have a pointcut so we assume it applies.
            return true;
        }
    }

/**
     * 判断当前增强器是否为本来能用的
     * */
    public static boolean canApply(Pointcut pc, Class<?> targetClass, boolean hasIntroductions) {
        Assert.notNull(pc, "Pointcut must not be null");
        if (!pc.getClassFilter().matches(targetClass)) {
            return false;
        }

        /**
         * 获取切点中的方法匹配器 TransactionAttributeSourcePointcut
         * 该切点在创建BeanFactoryTransactionAttributeSourceAdvisor的时候 创建了切点TransactionAttributeSourcePointcut
         * */
        MethodMatcher methodMatcher = pc.getMethodMatcher();
        if (methodMatcher == MethodMatcher.TRUE) {
            // No need to iterate the methods if we're matching any method anyway...
            return true;
        }

        /**
         * 判断方法匹配器是不是IntroductionAwareMethodMatcher
```

```
     * */
        IntroductionAwareMethodMatcher introductionAwareMethodMatcher = null;
        if (methodMatcher instanceof IntroductionAwareMethodMatcher) {
            introductionAwareMethodMatcher = (IntroductionAwareMethodMatcher) methodMatcher;
        }

    //获取当前类的实现接口类型
        Set<Class<?>> classes = new LinkedHashSet<Class<?>>(ClassUtils.getAllInterfacesForClassAsSet(targetClass)
        classes.add(targetClass);
        //循环上一步的接口类型
        for (Class<?> clazz : classes) {
           //获取接口的所有方法
            Method[] methods = ReflectionUtils.getAllDeclaredMethods(clazz);
            //循环我们接口中的方法
            for (Method method : methods) {
               //正在进行匹配的是methodMatcher.matches(method, targetClass)这个逻辑
                if ((introductionAwareMethodMatcher != null &&introductionAwareMethodMatcher.matches(method,
                        methodMatcher.matches(method, targetClass)) {
                    return true;
                }
            }
        }

        return false;
    }

    org.springframework.transaction.interceptor.TransactionAttributeSourcePointcut#matches

    public boolean matches(Method method, Class<?> targetClass) {
        if (targetClass != null && TransactionalProxy.class.isAssignableFrom(targetClass)) {
            return false;
        }
        //获取我们的事物源对象（在ProxyTransactionManagementConfiguration配置类配置的这里获取）
        TransactionAttributeSource tas = getTransactionAttributeSource();
        //从事物源对象中获取事物属性
        return (tas == null || tas.getTransactionAttribute(method, targetClass) != null);
    }


    /**
     * 获取事物属性对象
     * */
    public TransactionAttribute getTransactionAttribute(Method method, Class<?> targetClass) {
        if (method.getDeclaringClass() == Object.class) {
            return null;
        }

        //通过目标类和目标类的接口方法 拼接缓存key
        Object cacheKey = getCacheKey(method, targetClass);
        //去缓存中获取
        TransactionAttribute cached = this.attributeCache.get(cacheKey);
        if (cached != null) {
            //缓存中有 直接返回就可以了
            if (cached == NULL_TRANSACTION_ATTRIBUTE) {
                return null;
            }
            else {
                return cached;
            }
        }
        else {
            //计算事物属性.
```

```
                TransactionAttribute txAttr = computeTransactionAttribute(method, targetClass);
                //若事物属性为空.
                if (txAttr == null) {
                   //在缓存中标识 为事物方法
                    this.attributeCache.put(cacheKey, NULL_TRANSACTION_ATTRIBUTE);
                }
                else {
                    String methodIdentification = ClassUtils.getQualifiedMethodName(method, targetClass);
                    //为事物属性设置方法描述符号
                    if (txAttr instanceof DefaultTransactionAttribute) {
                        ((DefaultTransactionAttribute) txAttr).setDescriptor(methodIdentification);
                    }
                    if (logger.isDebugEnabled()) {
                        logger.debug("Adding transactional method '" + methodIdentification + "' with attribute: " + txA
                    }
                    //加入到缓存
                    this.attributeCache.put(cacheKey, txAttr);
                }
                return txAttr;
            }
        }


    /**
     * 计算事物属性
     * */
        protected TransactionAttribute computeTransactionAttribute(Method method, Class<?> targetClass) {
            //判断方法的修饰夫
            if (allowPublicMethodsOnly() && !Modifier.isPublic(method.getModifiers())) {
                return null;
            }

            //忽略cglib的代理
            Class<?> userClass = ClassUtils.getUserClass(targetClass);
            /**
             * method为接口中的方法,specificMethod为我们实现类方法
             * */
            Method specificMethod = ClassUtils.getMostSpecificMethod(method, userClass);
            // If we are dealing with method with generic parameters, find the original method.
            specificMethod = BridgeMethodResolver.findBridgedMethod(specificMethod);

            // 找我们【实现类】中的【方法】上的事物属性
            TransactionAttribute txAttr = findTransactionAttribute(specificMethod);
            if (txAttr != null) {
                return txAttr;
            }

          //【方法所在类】上有没有事物属性
            txAttr = findTransactionAttribute(specificMethod.getDeclaringClass());
            if (txAttr != null && ClassUtils.isUserLevelMethod(method)) {
                return txAttr;
            }

        【接口上的指定的方法】
            if (specificMethod != method) {
                // Fallback is to look at the original method.
                txAttr = findTransactionAttribute(method);
                if (txAttr != null) {
                    return txAttr;
                }
               【接口上】
                txAttr = findTransactionAttribute(method.getDeclaringClass());
```

```
                if (txAttr != null && ClassUtils.isUserLevelMethod(method)) {
                    return txAttr;
                }
            }

            return null;
        }


    /**
     * 从方法上找事物属性对象
     * */
        protected TransactionAttribute findTransactionAttribute(Method method) {
            return determineTransactionAttribute(method);
        }

        protected TransactionAttribute determineTransactionAttribute(AnnotatedElement element) {

            //获取方法上的注解
            if (element.getAnnotations().length > 0) {
                //事物注解解析器
                for (TransactionAnnotationParser annotationParser : this.annotationParsers) {
                    //解析我们的注解
                    TransactionAttribute attr = annotationParser.parseTransactionAnnotation(element);
                    if (attr != null) {
                        return attr;
                    }
                }
            }
            return null;
        }

        /**
         * 解析事物注解
         * */
        public TransactionAttribute parseTransactionAnnotation(AnnotatedElement element) {
            //解析@Transactional属性对象
            AnnotationAttributes attributes = AnnotatedElementUtils.getMergedAnnotationAttributes(
                    element, Transactional.class);
            if (attributes != null) {
                //真正的解析@Transactional属性
                return parseTransactionAnnotation(attributes);
            }
            else {
                return null;
            }
        }


    /**
     * 解析事物注解
     * */
        protected TransactionAttribute parseTransactionAnnotation(AnnotationAttributes attributes) {
            RuleBasedTransactionAttribute rbta = new RuleBasedTransactionAttribute();

        //传播行为
            Propagation propagation = attributes.getEnum("propagation");
            rbta.setPropagationBehavior(propagation.value());
            //隔离级别
            Isolation isolation = attributes.getEnum("isolation");
            rbta.setIsolationLevel(isolation.value());
            //事物超时
```

```
        rbta.setTimeout(attributes.getNumber("timeout").intValue());
        //判断是否为只读事物
        rbta.setReadOnly(attributes.getBoolean("readOnly"));
        //事物的名称吧
        rbta.setQualifier(attributes.getString("value"));

        List<RollbackRuleAttribute> rollbackRules = new ArrayList<RollbackRuleAttribute>();
        //事物回滚规则
        for (Class<?> rbRule : attributes.getClassArray("rollbackFor")) {
            rollbackRules.add(new RollbackRuleAttribute(rbRule));
        }

        //对哪个类进行回滚
        for (String rbRule : attributes.getStringArray("rollbackForClassName")) {
            rollbackRules.add(new RollbackRuleAttribute(rbRule));
        }
        //对哪些异常不回滚
        for (Class<?> rbRule : attributes.getClassArray("noRollbackFor")) {
            rollbackRules.add(new NoRollbackRuleAttribute(rbRule));
        }
        //对哪些类不回滚
        for (String rbRule : attributes.getStringArray("noRollbackForClassName")) {
            rollbackRules.add(new NoRollbackRuleAttribute(rbRule));
        }
        rbta.setRollbackRules(rollbackRules);

        return rbta;
    }
```

4.2.1）真正的创建代理对象

org.springframework.aop.framework.autoproxy.AbstractAutoProxyCreator#crea

```
    protected Object createProxy(
            Class<?> beanClass, String beanName, Object[] specificInterceptors, TargetSource targetSource) {

        //暴露代理对象
        if (this.beanFactory instanceof ConfigurableListableBeanFactory) {
            AutoProxyUtils.exposeTargetClass((ConfigurableListableBeanFactory) this.beanFactory, beanName, beanCl
        }

    //创建代理工厂
        ProxyFactory proxyFactory = new ProxyFactory();
        proxyFactory.copyFrom(this);

    //判断是cglib代理还是jdk代理
        if (!proxyFactory.isProxyTargetClass()) {
            if (shouldProxyTargetClass(beanClass, beanName)) {
                proxyFactory.setProxyTargetClass(true);
            }
            else {
                evaluateProxyInterfaces(beanClass, proxyFactory);
            }
        }

    //把合适的拦截器转为增强器
        Advisor[] advisors = buildAdvisors(beanName, specificInterceptors);
        proxyFactory.addAdvisors(advisors);
        proxyFactory.setTargetSource(targetSource);
```

```
        customizeProxyFactory(proxyFactory);

        proxyFactory.setFrozen(this.freezeProxy);
        if (advisorsPreFiltered()) {
            proxyFactory.setPreFiltered(true);
        }

    //真正的创建代理对象
        return proxyFactory.getProxy(getProxyClassLoader());
    }


    public AopProxy createAopProxy(AdvisedSupport config) throws AopConfigException {

        if (config.isOptimize() || config.isProxyTargetClass() || hasNoUserSuppliedProxyInterfaces(config)) {
            Class<?> targetClass = config.getTargetClass();
            if (targetClass == null) {
                throw new AopConfigException("TargetSource cannot determine target class: " +
                        "Either an interface or a target is required for proxy creation.");
            }
            //标识的是接口或者
            if (targetClass.isInterface() || Proxy.isProxyClass(targetClass)) {
                return new JdkDynamicAopProxy(config);
            }
            //创建cglib接口
            return new ObjenesisCglibAopProxy(config);
        }
        else {
           创建jdk代理
            return new JdkDynamicAopProxy(config);
        }
    }

    public Object getProxy(ClassLoader classLoader) {
        if (logger.isDebugEnabled()) {
            logger.debug("Creating JDK dynamic proxy: target source is " + this.advised.getTargetSource());
        }
        Class<?>[] proxiedInterfaces = AopProxyUtils.completeProxiedInterfaces(this.advised, true);
        findDefinedEqualsAndHashCodeMethods(proxiedInterfaces);
        return Proxy.newProxyInstance(classLoader, proxiedInterfaces, this);
    }
```

五:代理对象调用流程

5.1）org.springframework.aop.framework.JdkDynamicAopProxy#invoke

```
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        MethodInvocation invocation;
        Object oldProxy = null;
        boolean setProxyContext = false;

        TargetSource targetSource = this.advised.targetSource;
        Class<?> targetClass = null;
        Object target = null;

        try {

            Object retVal;
      //暴露代理对象
            if (this.advised.exposeProxy) {
```

```
                    // Make invocation available if necessary.
                    oldProxy = AopContext.setCurrentProxy(proxy);
                    setProxyContext = true;
                }

                // May be null. Get as late as possible to minimize the time we "own" the target,
                // in case it comes from a pool.
                target = targetSource.getTarget();
                if (target != null) {
                    targetClass = target.getClass();
                }

                //把增强器转为方法拦截器链条
                List<Object> chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);

              //拦截器链为空,直接通过反射进行调用
                if (chain.isEmpty()) {
            //通过反射进行调用
                    Object[] argsToUse = AopProxyUtils.adaptArgumentsIfNecessary(method, args);
                    retVal = AopUtils.invokeJoinpointUsingReflection(target, method, argsToUse);
                }
                else {
                    //创建反射方法调用对象
                    invocation = new ReflectiveMethodInvocation(proxy, target, method, args, targetClass, chain);
                  //通过方法拦截器进行拦截调用
                    retVal = invocation.proceed();
                }

                // Massage return value if necessary.
                Class<?> returnType = method.getReturnType();
                if (retVal != null && retVal == target &&
                        returnType != Object.class && returnType.isInstance(proxy) &&
                        !RawTargetAccess.class.isAssignableFrom(method.getDeclaringClass())) {
                    // Special case: it returned "this" and the return type of the method
                    // is type-compatible. Note that we can't help if the target sets
                    // a reference to itself in another returned object.
                    retVal = proxy;
                }
                else if (retVal == null && returnType != Void.TYPE && returnType.isPrimitive()) {
                    throw new AopInvocationException(
                            "Null return value from advice does not match primitive return type for: " + method);
                }
                return retVal;
            }
            finally {
                if (target != null && !targetSource.isStatic()) {
                    // Must have come from TargetSource.
                    targetSource.releaseTarget(target);
                }
                if (setProxyContext) {
                    // Restore old proxy.
                    AopContext.setCurrentProxy(oldProxy);
                }
            }
        }
```

5.2)org.springframework.aop.framework.ReflectiveMethodInvocation#proceed

```
    public Object proceed() throws Throwable {
        //当前下标从-1开始,若当前索引值=执行到最后一个拦截器的下标,就执行目标方法
        if (this.currentInterceptorIndex == this.interceptorsAndDynamicMethodMatchers.size() - 1) {
```

```
            return invokeJoinpoint();
        }

    //获取我们的方法拦截器(TransactionInterceptor)
        Object interceptorOrInterceptionAdvice =
                this.interceptorsAndDynamicMethodMatchers.get(++this.currentInterceptorIndex);
        if (interceptorOrInterceptionAdvice instanceof InterceptorAndDynamicMethodMatcher) {
            // Evaluate dynamic method matcher here: static part will already have
            // been evaluated and found to match.
            InterceptorAndDynamicMethodMatcher dm =
                    (InterceptorAndDynamicMethodMatcher) interceptorOrInterceptionAdvice;
            if (dm.methodMatcher.matches(this.method, this.targetClass, this.arguments)) {
                return dm.interceptor.invoke(this);
            }
            else {
                // Dynamic matching failed.
                // Skip this interceptor and invoke the next in the chain.
                return proceed();
            }
        }
        else {
      //事务拦截器进行调用
            return ((MethodInterceptor) interceptorOrInterceptionAdvice).invoke(this);
        }
    }
```

5.2）org.springframework.transaction.interceptor.TransactionInterceptor#invoke(**事务拦截器进行调用**)

```
    public Object invoke(final MethodInvocation invocation) throws Throwable {
      //获取代理对象的目标class
        Class<?> targetClass = (invocation.getThis() != null ? AopUtils.getTargetClass(invocation.getThis()) : null);

      //使用事务调用
        return invokeWithinTransaction(invocation.getMethod(), targetClass, new InvocationCallback() {
            //从这里触发调用目标方法的
            public Object proceedWithInvocation() throws Throwable {
                return invocation.proceed();
            }
        });
    }
```

5.3)org.springframework.transaction.interceptor.TransactionAspectSupport#inv**务调用)**

```
    protected Object invokeWithinTransaction(Method method, Class<?> targetClass, final InvocationCallback invocatior
            throws Throwable {

        //通过@EnableTransactionManager 到入了TransactionAttributeSource  可以获取出事务属性对象
        final TransactionAttribute txAttr = getTransactionAttributeSource().getTransactionAttribute(method, targetClas
        //获取工程中的事务管理器
        final PlatformTransactionManager tm = determineTransactionManager(txAttr);
        //获取我们需要切入的方法(也就是我们标识了@Transactional注解的方法)
        final String joinpointIdentification = methodIdentification(method, targetClass, txAttr);

    //再这里我们只看我们常用的事务管理器,很明显我们不会配置CallbackPreferringPlatformTransactionManager事务管理器
        if (txAttr == null || !(tm instanceof CallbackPreferringPlatformTransactionManager)) {
          //判断有没有必要开启事务
            TransactionInfo txInfo = createTransactionIfNecessary(tm, txAttr, joinpointIdentification);
            Object retVal = null;
```

```
            try {
                //调用我们的目标方法
                retVal = invocation.proceedWithInvocation();
            }
            catch (Throwable ex) {
                //抛出异常进行回顾
                completeTransactionAfterThrowing(txInfo, ex);
                throw ex;
            }
            finally {
                //清除事务信息
                cleanupTransactionInfo(txInfo);
            }
            //提交事务
            commitTransactionAfterReturning(txInfo);
            return retVal;
        }

    }
```

5.3.1) org.springframework.transaction.interceptor.TransactionAspectSupport#createT

```
    protected TransactionInfo createTransactionIfNecessary(
            PlatformTransactionManager tm, TransactionAttribute txAttr, final String joinpointIdentification) {

        //把事务属性包装为
        if (txAttr != null && txAttr.getName() == null) {
            txAttr = new DelegatingTransactionAttribute(txAttr) {
                @Override
                public String getName() {
                    return joinpointIdentification;
                }
            };
        }

        TransactionStatus status = null;
        if (txAttr != null) {
            if (tm != null) {
                //获取一个事务状态
                status = tm.getTransaction(txAttr);
            }
            else {
                if (logger.isDebugEnabled()) {
                    logger.debug("Skipping transactional joinpoint [" + joinpointIdentification +
                            "] because no transaction manager has been configured");
                }
            }
        }
        //准备事务信息
        return prepareTransactionInfo(tm, txAttr, joinpointIdentification, status);
    }
```

5.3.2)org.springframework.transaction.support.AbstractPlatformTransactionMa

```
    public final TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException {
        1:)先去尝试开启一个事务
        Object transaction = doGetTransaction();
```

```
        // Cache debug flag to avoid repeated checks.
        boolean debugEnabled = logger.isDebugEnabled();
    //传入进来的事务定义为空
        if (definition == null) {
            //使用系统默认的
            definition = new DefaultTransactionDefinition();
        }

    2:)//判断是否存在事务（若存在事务，在这边直接返回不走下面的处理了）
        if (isExistingTransaction(transaction)) {
            // Existing transaction found -> check propagation behavior to find out how to behave.
            return handleExistingTransaction(definition, transaction, debugEnabled);
        }

        3:)判读事务超时
        if (definition.getTimeout() < TransactionDefinition.TIMEOUT_DEFAULT) {
            throw new InvalidTimeoutException("Invalid transaction timeout", definition.getTimeout());
        }

        //不存在事务,需要在这边判断(PROPAGATION_MANDATORY 标识要求当前允许的在事务中，但是第二步进行判断之后 说
        if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_MANDATORY) {
            throw new IllegalTransactionStateException(
                    "No existing transaction found for transaction marked with propagation 'mandatory'");
        }
        //PROPAGATION_REQUIRED
        //PROPAGATION_REQUIRES_NEW
        //PROPAGATION_NESTED
        else if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_REQUIRED ||
                definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_REQUIRES_NEW ||
                definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_NESTED) {
            //挂起当前事务,但是当前是没有事务的
            SuspendedResourcesHolder suspendedResources = suspend(null);
            if (debugEnabled) {
                logger.debug("Creating new transaction with name [" + definition.getName() + "]: " + definition);
            }
            try {
                boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION_NEVER);
                //创建一个新的事物状态
                DefaultTransactionStatus status = newTransactionStatus(
                        definition, transaction, true, newSynchronization, debugEnabled, suspendedResources);
                //开启一个事物
                doBegin(transaction, definition);
                //准备事物同步
                prepareSynchronization(status, definition);
                return status;
            }
            catch (RuntimeException ex) {
                resume(null, suspendedResources);
                throw ex;
            }
            catch (Error err) {
                resume(null, suspendedResources);
                throw err;
            }
        }
        else {
           //创建一个空的事物.
            if (definition.getIsolationLevel() != TransactionDefinition.ISOLATION_DEFAULT && logger.isWarnEnabled())
                logger.warn("Custom isolation level specified but no actual transaction initiated; " +
                        "isolation level will effectively be ignored: " + definition);
            }
```

```
                boolean newSynchronization = (getTransactionSynchronization() == SYNCHRONIZATION_ALWAYS);
                return prepareTransactionStatus(definition, null, true, newSynchronization, debugEnabled, null);
            }
        }


==================================================代码1处中的代码=================
        /**
         * 第一次进来的时候,是没有事务持有对象
         * */
        protected Object doGetTransaction() {
            //创建一个数据库事务管理器
            DataSourceTransactionObject txObject = new DataSourceTransactionObject();
            //设置一个事务保存点
            txObject.setSavepointAllowed(isNestedTransactionAllowed());
            //从事务同步管理器中获取连接持有器
            ConnectionHolder conHolder =
                    (ConnectionHolder) TransactionSynchronizationManager.getResource(this.dataSource);
            //把持有器设置到对象中
            txObject.setConnectionHolder(conHolder, false);
            //返回一个事务对象
            return txObject;
        }

    //第一次进来不会走这个逻辑
        protected boolean isExistingTransaction(Object transaction) {
            //获取事务对象中的持有器
            DataSourceTransactionObject txObject = (DataSourceTransactionObject) transaction;
            //持有器不为空且  有事务激活
            return (txObject.hasConnectionHolder() && txObject.getConnectionHolder().isTransactionActive());
        }

======================handleExistingTransaction================================



/**
 * 第一次调用的时候
 * */
protected void doBegin(Object transaction, TransactionDefinition definition) {

            DataSourceTransactionObject txObject = (DataSourceTransactionObject) transaction;
            Connection con = null;

            try {
                //第一次进来，事务持有器中是没有对象的,所以我们需要自己手动的设置进去
                if (!txObject.hasConnectionHolder() ||
                        txObject.getConnectionHolder().isSynchronizedWithTransaction()) {
                    //获取一个数据库连接
                    Connection newCon = this.dataSource.getConnection();
                    if (logger.isDebugEnabled()) {
                        logger.debug("Acquired Connection [" + newCon + "] for JDBC transaction");
                    }
                    //把数据库连接封装为一个持有器对象并且设置到事务对象中
                    txObject.setConnectionHolder(new ConnectionHolder(newCon), true);
                }

            //开始同步标志
                    txObject.getConnectionHolder().setSynchronizedWithTransaction(true);

                    con = txObject.getConnectionHolder().getConnection();
            //获取事务的隔离级别
```

```
                Integer previousIsolationLevel = DataSourceUtils.prepareConnectionForTransaction(con, definition);
                txObject.setPreviousIsolationLevel(previousIsolationLevel);

        /**
         * 关闭事务自动提交
         * */
                if (con.getAutoCommit()) {
                    txObject.setMustRestoreAutoCommit(true);
                    if (logger.isDebugEnabled()) {
                        logger.debug("Switching JDBC Connection [" + con + "] to manual commit");
                    }
                    con.setAutoCommit(false);
                }

        //判断事务是不是为只读的事务
                prepareTransactionalConnection(con, definition);
                //设置事务激活
                txObject.getConnectionHolder().setTransactionActive(true);

                int timeout = determineTimeout(definition);
                if (timeout != TransactionDefinition.TIMEOUT_DEFAULT) {
                    txObject.getConnectionHolder().setTimeoutInSeconds(timeout);
                }

                //把数据源和事务持有器保存到事务同步管理器中
                if (txObject.isNewConnectionHolder()) {
                    TransactionSynchronizationManager.bindResource(getDataSource(), txObject.getConnectionHolder())
                }
            }

            catch (Throwable ex) {
                if (txObject.isNewConnectionHolder()) {
                   //抛出异常,释放资源
                    DataSourceUtils.releaseConnection(con, this.dataSource);
                    txObject.setConnectionHolder(null, false);
                }
                throw new CannotCreateTransactionException("Could not open JDBC Connection for transaction", ex);
            }
        }

============================      prepareSynchronization(status, definition);把当前的事务设置到同步管理器中(
    protected void prepareSynchronization(DefaultTransactionStatus status, TransactionDefinition definition) {
        if (status.isNewSynchronization()) {
            //设置事务激活
            TransactionSynchronizationManager.setActualTransactionActive(status.hasTransaction());
            //设置隔离级别
            TransactionSynchronizationManager.setCurrentTransactionIsolationLevel(
                    definition.getIsolationLevel() != TransactionDefinition.ISOLATION_DEFAULT ?
                            definition.getIsolationLevel() : null);
            //设置只读书屋
            ransactionSynchronizationManager.setCurrentTransactionReadOnly(definition.isReadOnly());
            //设置事务的名称
            TransactionSynchronizationManager.setCurrentTransactionName(definition.getName());
            TransactionSynchronizationManager.initSynchronization();
        }
    }


==================================      //准备事务信息 prepareTransactionInfo(tm, txAttr, joinpoint
protected TransactionInfo prepareTransactionInfo(PlatformTransactionManager tm,
            TransactionAttribute txAttr, String joinpointIdentification, TransactionStatus status) {
```

```
//把事务管理器，事务属性,连接点信息封装成为TransactionInfo
        TransactionInfo txInfo = new TransactionInfo(tm, txAttr, joinpointIdentification);
        if (txAttr != null) {
                // We need a transaction for this method...
                if (logger.isTraceEnabled()) {
                        logger.trace("Getting transaction for [" + txInfo.getJoinpointIdentification() + "]");
                }
              //设置事务状态
                txInfo.newTransactionStatus(status);
        }
        else {
                // The TransactionInfo.hasTransaction() method will return false. We created it only
                // to preserve the integrity of the ThreadLocal stack maintained in this class.
                if (logger.isTraceEnabled())
                        logger.trace("Don't need to create transaction for [" + joinpointIdentification +
                                        "]: This method isn't transactional.");
        }

//把事务信息绑定到当前线程上去
        txInfo.bindToThread();
        return txInfo;
}
```